OA&M API for Linux Operating Systems

Programming Guide

August 2005

05-1850-004

Publication Date: August 2005

Document Number: 05-1850-004

Intel Converged Communications, Inc.
1515 Route 10
Parsippany, NJ 07054

For **Technical Support**, visit the Intel Telecom Support Resources website at:
http://developer.intel.com/design/telecom/support

For **Products and Services Information**, visit the Intel Telecom Products website at:
http://www.intel.com/design/network/products/telecom

For **Sales Offices** and other contact information, visit the Where to Buy Intel Telecom Products page at:
http://www.intel.com/buy/wtb/wtb1028.htm

Contents

Figures

Revision History

This revision history summarizes the changes made in each published version of this document.

Document No.	Publication Date	Description of Revisions
05-1850-004	August 2005	General Guidelines section: Added information about freeing memory that is allocated for object pointers.
05-1850-003	March 2005	Application Development Guidelines chapter: Added section about High Availability development guidelines. High Availability is a feature of cPCI boards. The Intel® Dialogic® System Release 6.1 Feature Release 2 for Linux supports PCI and cPCI boards.
05-1850-002	November 2004	Product Description chapter: Noted that the CTPLATFORMVERSIONINFO data structure has been updated for the current release. Noted that Springware architecture boards only generate events on the NETWORK_ALARM_CHANNEL. DM3 architecture boards generate events on all four event service channels. Application Development Guidelines chapter: Removed section about High Availability development guidelines. High Availability is a feature of cPCI boards. The Intel® Dialogic® System Release 6.1 Feature Release 1 for Linux does not support cPCI boards.
05-1850-001	September 2002	Initial version of document.

About This Publication

The following topics provide information about this publication:

Purpose

This publication provides guidelines for building applications using the Operations, Administration and Maintenance (OA&M) API. The OA&M API is used to create system management applications (for example, a customized CT bus clocking daemon).

This publication is a companion guide to the *OA&M API for Linux Operating Systems Library Reference* that provides details on the classes, functions, error codes and events in the OA&M library.

Intended Audience

This publication is written for the following audience:

- Distributors
- System Integrators
- Toolkit Developers
- Independent Software Vendors (ISVs)
- Value Added Resellers (VARs)
- Original Equipment Manufacturers (OEMs)

How to Use This Publication

Refer to this publication after you have installed the hardware and the system software that includes the OA&M API library.

This publication assumes that you are familiar with the Linux operating system and the C++ programming language. If you are developing a CT Bus management application, you should also have a thorough understanding of CT Bus clocking concepts (Primary, Secondary and Network Reference clock masters).

The information in this guide is organized as follows:

- Chapter 1, "Product Description" provides an overview of the OA&M API.
- Chapter 2, "Event Handling" describes how to register with the OA&M event notification framework and handle OA&M events.
- Chapter 3, "Error Handling" describes the exception handling capabilities provided by the OA&M API.
- Chapter 4, "Application Development Guidelines" provides guidelines when developing applications from the OA&M API.
- Chapter 5, "Building Applications" provides guidelines for building applications with the OA&M API.

Related Information

Refer to the following documents and Web sites for more information:

- *OA&M API for Linux Operating Systems Library Reference*
- *Intel® DM3 Architecture Products on Linux Configuration Guide*
- *Intel® Springware Architecture Products on Linux Configuration Guide*
- *Intel® Dialogic® System Release on Linux Administration Guide*
- *High Availability for Linux Operating System Demo Guide*
- *Intel NetStructure® IPT Series on Linux Configuration Guide*
- *System Release Guide*
- *System Release Update*
- *http://developer.intel.com/design/telecom/support/* (for technical support)
- *http://www.ectf.org/* (for CT Bus clocking specifications)

Product Description 1

This chapter describes the OA&M API. The following topics are included:

1.1 OA&M API Overview

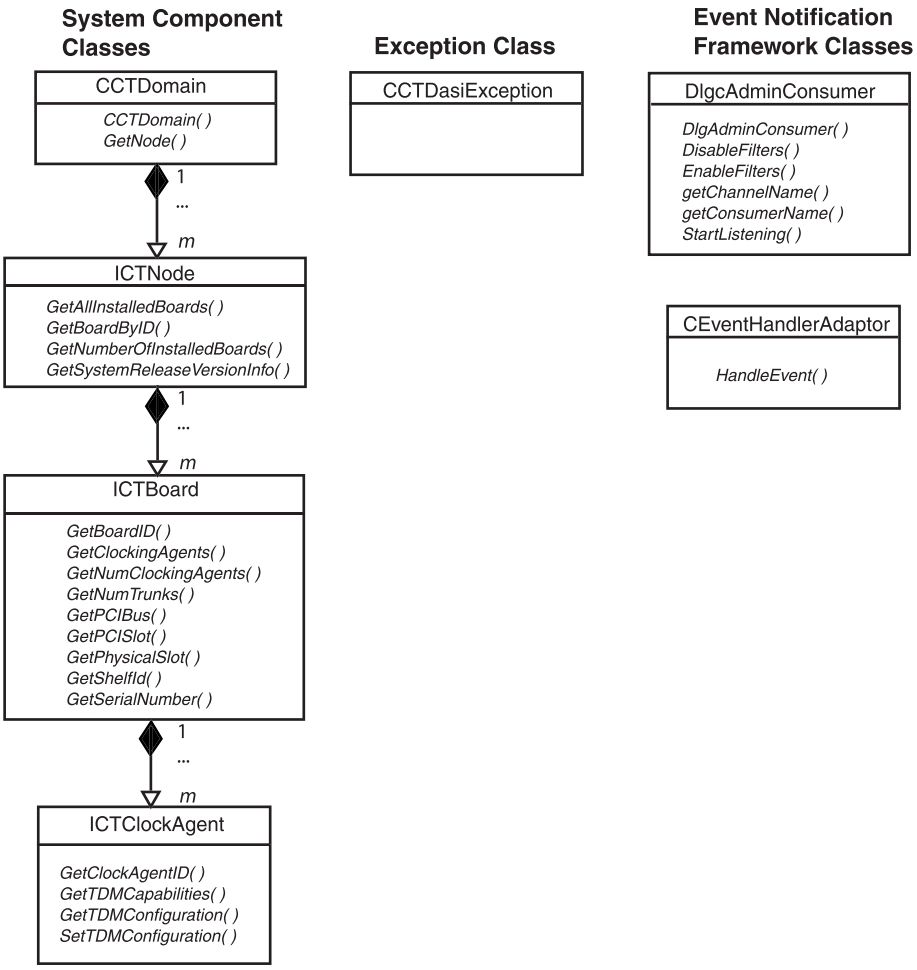
The classes and functions of the OA&M API support the development of applications that facilitate the Operation, Administration, and Maintenance (OA&M) of Intel® Dialogic systems. For example, you can create a system management application that monitors CT Bus clocking fault events and automatically performs clock reference (Primary, Secondary, or Network Reference) recovery without user intervention.

The classes in the OA&M API can be divided into the following three categories:

- System Component classes (CCTDomain, ICTNode, ICTBoard, ICTClockAgent). The System Component classes allow you to instantiate objects that represent the various components of your Intel® Dialogic system (domain, node, boards, clocking agents). After an object has been instantiated, you can use the member functions of its class to programmatically obtain, and in some cases change, the object's attributes.
- Exception class (CCTDasiException). The exception class defines the exception objects that can be generated by functions in the CCTDomain, ICTNode, ICTBoard, and ICTClockAgent classes.
- Event notification framework classes (DlgAdminConsumer, CEventHandlerAdaptor). The event notification framework classes allow you to instantiate objects that receive and process OA&M events.

Figure 1 depicts the names and member functions of each class included in the OA&M API library.

Figure 1. OA&M API Classes



1.2 Features

The features available through the OA&M API include the following:

- Get version information (major release number, release build number, service pack number, feature pack number, service update number, feature release number etc.) about the Intel® Dialogic system software that is installed on a node (ICTNode class).

  Note: The CTPLATFORMVERSIONINFO data structure has been updated as part of the current system software release. To incorporate the updates, you must recompile any existing application that uses the CTPLATFORMVERSIONINFO data structure or the **ICTNode::GetSystemReleaseVersionInfo( )** function. Refer to the *OA&M API*

for Linux Operating Systems Library Reference for information about the CTPLATFORMVERSIONINFO data structure and the **ICTNode::GetSystemReleaseVersionInfo( )** function.

- Get physical identification information (serial number, PCI slot number, number of network interface trunks, etc.) about individual DM3 architecture or Springware architecture boards (ICTBoard class).
- Get the Addressable Unit Identifier (AUID) of an individual DM3 architecture or Springware architecture board (ICTBoard class).
- Get the AUID of a board's clocking agent (ICTClockAgent class).
- Get the TDM bus capabilities of a board's clocking agent (ICTClockAgent class).
- Get the TDM bus configuration of a board's clocking agent (ICTClockAgent class).
- Set the TDM bus configuration of a board's clocking agent (ICTClockAgent class).
- Receive asynchronous OA&M events from the event notification framework (DlgAdminConsumer class).
- Instantiate event handler objects. Event handler objects are invoked when DlgAdminConsumer objects receive events from the event notification framework (CEventHandlerAdaptor class).

1.3 OA&M API Class Hierarchy

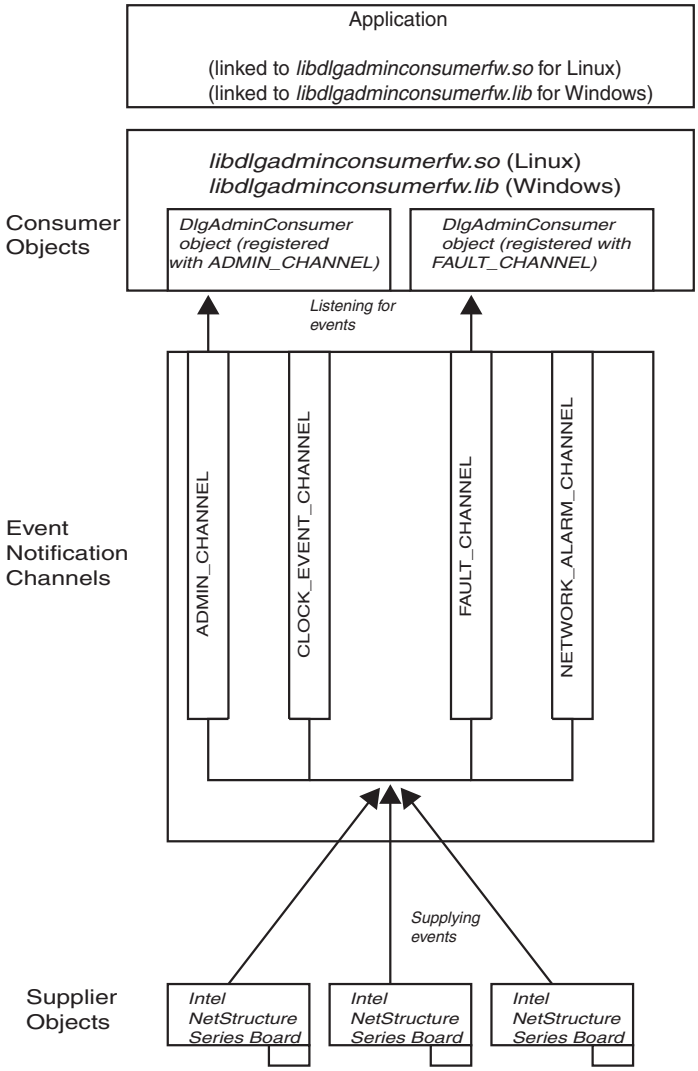
The System Component classes of the OA&M API (CCTDomain, ICTNode, ICTBoard, and ICTClockAgent) are organized into a hierarchy as shown in Figure 1, "OA&M API Classes", on page 10. Since the CCTDomain class resides at the top of the class hierarchy, each application built from the OA&M API must first instantiate a CCTDomain object as an entry point into the OA&M system. You must then adhere to the class hierarchy as you instantiate other System Component objects. For example, since the ICTClockAgent class resides at the bottom of the OA&M API class hierarchy, you must instantiate and query all other System Component objects (CCTDomain, ICTNode, ICTBoard) until you reach the ICTClockAgent class.

Refer to Chapter 4, "Application Development Guidelines" for procedural information about instantiating CCTDomain, ICTNode, ICTBoard, and ICTClockAgent objects.

1.4 Event Notification Framework

The event notification framework is the Intel® Dialogic subsystem for sending asynchronous OA&M events to applications. The framework is implemented using supplier objects, consumer objects and event notification channels as shown in Figure 2:

Figure 2. Event Notification Framework



Note: The CLOCK_EVENT_CHANNEL and NETWORK_ALARM_CHANNEL are idle in the figure shown above. These two channels are available, but do not have consumer objects registered to listen for events.

Intel® Dialogic system software components, such as device drivers and firmware, are the supplier objects. They generate events that are broadcast to consumer objects via the event notification channels. The DlgAdminConsumer class allows you to instantiate consumer objects and register them to receive events from one of the event notification channels. The CEventHandlerAdaptor class allows you to instantiate user-defined event handler objects that are invoked when DlgAdminConsumer objects receive events.

The framework contains the following event notification channels, each of which carries specific types of events:

- ADMIN_CHANNEL
- CLOCK_EVENT_CHANNEL
- FAULT_CHANNEL
- NETWORK_ALARM_CHANNEL

Notes: ***1.*** DM3 architecture boards generate events on all four channels shown above. Springware architecture boards only generate NETWORK_ALARM_CHANNEL events.

2. Each DlgAdminConsumer object can only monitor one event notification channel for incoming events.

Refer to Chapter 2, "Event Handling" for more information about the event notification framework.

Event Handling 2

This chapter provides information about receiving and handling asynchronous events that are transmitted via the event notification framework.

For your application to receive events from the event notification framework you must follow these steps:

1. Define and implement a class that is derived from the CEventHandlerAdaptor class.
2. Provide an implementation of the **CEventHandlerAdaptor::HandleEvent( )** function.
3. Define an array of filters for use by the DlgAdminConsumer object using DlgEventService::AdminConsumer::FilterCallbackAssoc. Each DlgAdminConsumer object references an array of filters. The elements in the array determine the following:
   - the event handler that is associated with the DlgAdminConsumer object. The implementation of the **CEventHandlerAdaptor::HandleEvent( )** function is called when one of the events that is included in the array is received.
   - the client data that is returned to the application after the associated event handler object has been invoked.
   - the events that are allowed to pass to the DlgAdminConsumer object. If an event does not have an element in the filter array, the event is discarded and the DlgAdminConsumer object will not receive it.
   - whether the event filter is enabled or disabled.
4. Use the **DlgAdminConsumer::DlgAdminConsumer( )** function to instantiate a consumer object. A pointer to the array initialized in step 3 is used as a parameter for the function.
5. Call the **DlgAdminConsumer::StartListening( )** function so that the consumer object can begin monitoring its associated event notification channel for events. When the **DlgAdminConsumer::StartListening( )** function is called, the consumer object creates and runs in its own thread, allowing it to monitor its associated event notification channel without blocking the main application thread.

When a DlgAdminConsumer object receives an event through its associated event notification channel, it compares the event's msgId field to its filter array. If a matching filter is found, the associated event handler object is invoked.

Note: When a DlgAdminConsumer object receives more than one event that is associated with the same event handler object, the DlgAdminConsumer object must return from the **CEventHandlerAdaptor::HandleEvent( )** function before it can process the next event.

Error Handling 3

This chapter describes the error handling capabilities provided by the OA&M API.

When an error occurs during execution of a function in one of the System Component classes (CCTDomain, ICTNode, ICTBoard or ICTClockAgent), an exception is thrown. Exceptions can be caught and handled by including standard C++ try/catch blocks throughout your application code.

The following sample try/catch block catches and handles an exception that is thrown when the application attempts to use an incorrect parameter ("128.0.0.1" instead of "127.0.0.1") for the **CCTDomain::GetNode( )** function:

```
//processing....

...

//Initiate and catch DLG_INVALID_SERVER exception
   try
      {
      cout<< "Initiating DLG_INVALID_SERVER exception:"<<endl;
      //Attempt to get node from domain using incorrect default parameter. Exception thrown.
      pNode = pDomain->GetNode("128.0.0.1");
      }
   catch (CCTDasiException e)
      {
      cerr<<"Exception: "<<endl;
      cerr<<"Error Value: "<<hex<<e.m_dlgSysError<<endl;
      cerr<<"Error String: "<<e.m_errDesc<<endl;
      }

...

//continue processing
```

The CCTDasiException class defines the exceptions that can be thrown by the CCTDomain, ICTNode, ICTBoard or ICTClockAgent member functions.

For a complete list of error codes that can be included in CCTDasiException objects, refer to the *OA&M API for Linux Operating Systems Library Reference*.

Application Development Guidelines 4

This chapter provides information about developing applications using the OA&M API. It includes the following sections:

4.1 General Guidelines

The OA&M API contains four System Component classes:

- CCTDomain
- ICTNode
- ICTBoard
- ICTClockAgent

When an object is created from any one of these classes, the class member functions can be invoked to provide information about the created object. However, classes in the OA&M API are arranged in a hierarchy as shown in Figure 1, "OA&M API Classes", on page 10. Only CCTDomain objects can be directly instantiated. You must perform the following procedure to instantiate System Component objects in the OA&M API:

1. Use the **CCTDomain::CCTDomain( )** class constructor to instantiate an instance of the CCTDomain class.
2. Use the **CCTDomain::GetNode( )** function to get a pointer to an instance of the ICTNode class. The application is responsible for freeing the memory that is allocated for the pointer to the ICTNode object.
3. Use the **ICTNode::GetAllInstalledBoards( )** function to get a pointer to a C++ Standard Template Library (STL) list of pointers to ICTBoard objects. The application is responsible for freeing the memory that is allocated for the list of pointers to ICTBoard objects.
4. Walk the returned board list and use the **ICTBoard::GetClockingAgents( )** function to get a pointer to an STL list of pointers to ICTClockAgent objects. The application is responsible for freeing the memory that is allocated for the list of pointers to ICTClockAgent objects.

You can then use any of the following functions to get/set the TDM bus information of any one of the ICTClockAgent objects in the returned STL list:

- **ICTClockAgent::GetTDMBusCapabilities( )**
- **ICTClockAgent::GetTDMBusConfiguration( )**

- **ICTClockAgent::SetTDMBusConfiguration( )**

4.2 Designing CT Bus Clocking Applications

If you are using any of ICTClockAgent class member functions (for example, to develop a customized system clocking daemon) you must disable the clocking daemon that is included with the Intel® Dialogic system software by editing the **ClockDaemonMode** parameter in the *dlgsys.cfg* file. Refer to either of the following documents for more information about disabling the **ClockDaemonMode** parameter via the *dlgsys.cfg* file:

- *Intel® DM3 Architecture Products on Linux Configuration Guide*
- *Intel® Springware Architecture Products on Linux Configuration Guide*
- *Intel® Dialogic® System Release on Linux Administration Guide*
- *Intel NetStructure® IPT Series on Linux Configuration Guide*

4.3 Designing High Availability into Applications

This section provides instructions for using various Intel® Dialogic libraries to develop applications that support peripheral hot swap. The board replacement could be driven by an application (for example, if statistics are showing a degradation of service) or could be driven by an operator (for example, periodic hardware maintenance).

The following procedure provides information on developing applications that support Basic Hot Swap:

1. Register your application to receive events from the OA&M event notification framework according to the procedure outlined in Chapter 2, “Event Handling”.
2. When a CP/SP fault, CT Bus clocking fault or network alarm event is transmitted via the event notification framework, the event’s payload contains the AUID of the board that generated the event.
3. Once the application has a board’s AUID, use the **SRLGetVirtualBoardsOnPhysicalBoard( )** and **SRLGetSubDevicesOnVirtualBoard( )** functions to retrieve the list of virtual devices (“dxxx”, “dti” etc.) on a board. This allows the application to cross-reference physical boards with virtual devices. For more information about these functions, see the *Standard Runtime Library API for Linux and Windows Operating Systems Library Reference*.

   Note: As a general HA application rule, you should design your application to accommodate the dynamic removal/insertion of virtual devices. Do not include permanent references to virtual device names within your application; instead, depend on the SRL device mapper functions to get virtual device names.
4. The Standard Runtime Library functions can be used to determine the virtual devices on the board with the specified AUID, and the devices can be closed using **dx_close( )**, **dt_close( )**, etc.
5. When all virtual devices on a board have been closed, you can call the `stopbrd` utility to stop the board.

6. When the `stopbrd` utility successfully stops the board, use the `removebrd` utility to notify the system that the board is being removed.
7. Physically remove the board from the chassis when the board's Out of Service (Blue) LED lights.
8. Insert a new board into the chassis, configure it according to the procedures in the appropriate Configuration Guide and invoke the `startbrd` utility.
9. When a board has been started, a DLGC_EVT_BLADE_STARTED event is transmitted on the OA&M event notification framework's ADMIN_CHANNEL. Use the AUID from the event's payload, along with the Standard Runtime Library functions and virtual device functions (**dx_open( )**, **dt_open( )**, etc.) to determine the virtual devices that are on the newly inserted board and open the board's virtual devices.
10. Once the board's virtual devices are opened, the board can be used by your application.

Refer to the *System Release Administration Guide* for information about the `stopbrd`, `removebrd`, and `startbrd` utilities.

Refer to the *High Availability for Linux Operating System Demo Guide* for information on executing the High Availability demo programs included with the Intel® Dialogic® System Software.

Refer to the *Intel® DM3 Architecture Products on Linux Configuration Guide* or the *Intel NetStructure® IPT Series on Linux Configuration Guide* for complete information about configuring the boards.

Building Applications 5

This chapter provides general information for building applications that use the OA&M library. The following topics are included in this chapter:

5.1 Compiling and Linking

An application that uses the OA&M API library must include references to the OA&M API header files and must include the appropriate library files. This information is provided in the following topics:

- Include Files
- Required Libraries

5.1.1 Include Files

The following header file contains equates and #include directives that are required by **all** applications that uses the OA&M library:

dasi.h
primary OA&M API header file

The following header files are required by applications that receive and process events from the OA&M event notification framework:

dlgadminconsumer.h
defines the DlgAdminConsumer class

dlgadminmsg.h
defines the CEventHandlerAdaptor class

dlgadminevents.h
includes the payload format for ADMIN_CHANNEL events

dlgcclockingevents.h
includes the payload format for CLOCK_EVENT_CHANNEL events

dlgceventbasedef.h
provides the definitions of all event channels

dlgcprocfaultevents.h
includes the payload format for FAULT_CHANNEL events

dlgcnetworkalarmevents.h
includes the payload format for NETWORK_ALARM_CHANNEL events

dlgeventproxydef.h
defines the generic data structure for OA&M events

Note: User-defined header files that provide an implementation of the CEventHandlerAdaptor class (i.e., *adminhandler.h)* and override its **CEventHandlerAdaptor::HandleEvent( )** function also need to be included in any application that receives and processes OA&M events.

5.1.2 Required Libraries

The following library file must be linked to **all** applications that use the OA&M API:

libdlgdasi.so
primary OA&M API shared object file

The following library file must be linked to applications that receive and process events from the OA&M event notification framework:

libdlgadminconsumerfw.so
shared object file that contains event consumer objects

5.2 Variables for Compiling and Linking

In System Release 6.0 and later, the following variables are included to provide a standardized way of referencing the directories that contain header files and shared objects:

INTEL_DIALOGIC_INC
variable that points to the directory where header files are stored.

INTEL_DIALOGIC_LIB
variable that points to the directory where shared library files are stored.

These variables are automatically set at login and should be used in compiling and linking commands. The following is an example of a compiling and linking command that uses these variables:

```
cc -I${INTEL_DIALOGIC_INC} -o myapp myapp.c -L${INTEL_DIALOGIC_LIB} -lgc
```

Note: It is strongly recommended that developers begin using these variables when compiling and linking applications since they will be required in future releases. The name of the variables will remain constant, but the values may change in future releases.

Index